Software Risk Management From a System Perspective

George Holt
AdaRose Inc.

Software development can be fraught with frustration. Too often, we treat hardware risks and software risks as separate entities. Staying focused on the basics of risk management at the system level, from the get-go, is an essential part of minimizing risks and ensuring the success of even the most challenging and complex development projects. This article stresses the importance of managing risk from a system perspective by providing concrete examples of how one company applied the fundamentals of risk management to a military tactical system developed under less than ideal conditions.

Developing software can be challenging and rewarding but seldom easy. Developing software with a floating hardware baseline can be quite difficult. Add to this the commensurate development of test tools, simulators, and emulators by third parties, and then place schedule and cost constraints on the entire project, and you challenge even the best and the brightest.

Although each project entails unique demands, challenges, and problems, if we fail to predict and prevent risks from a system perspective it can lead to costly delays, increased stress on team members, a lesser product – even project failure.

This article stresses the importance of managing risk from a system perspective. It provides concrete examples of how one company, AdaRose Inc., applied the fundamentals of risk management to a military tactical system developed under less than ideal conditions such as those described above.

The Task at Hand

AdaRose engineers had prior experience developing software that resulted in the first tactical weapon system to run on a common PC architecture using a commercial operating system. The current task was to port this software to a new hardware architecture (still PC-based) that incorporated three single-board computers providing navigation, command-and-control, situational awareness, and real-time diagnostics.

Along with AdaRose, the Integrated Product Team (IPT) consisted of a major defense contractor, a small business hardware manufacturer, the Army end user, Army research and development specialists, and the Army product manager. All members of the IPT were experienced professionals with in-depth knowledge of the weapon system from both a functional and operational perspective. The software consisted of 230,000 lines of Ada code with specialized modules and drivers written in other high order languages.

The Solution

Although each project has its own requirements, the fundamentals of effective risk management at the system level remain the same. By identifying risks and developing solutions before and during the development process, you maximize the team's efficiency and the quality of the finished product.

"... the fundamentals of effective risk management at the system level remain the same. By identifying risks and developing solutions before and during the development process, you maximize the team's efficiency and the quality of the finished product."

I would like to start off by referring the reader to one of my earlier articles, "Risk Management Fundamentals in Software Development" published in the August 2000 issue of CrossTalk [1]. It describes how to implement an effective software risk management program. The fundamentals in that article can be applied, at the system level, to this military tactical system developed under less than ideal conditions.

Identifying the Risks

From the get-go, we were informed that this project would have significant risk drivers, i.e., (1) it would have cost and schedule constraints, (2) it would require software development before the hardware was built, and (3) tools such as lab simulators and emulators would have to be developed commensurate with the tactical software development. What was initially perceived as a straightforward port of software to a new hardware environment turned out to be a nontrivial undertaking.

System Level Risks

The challenge on this project soon became evident. On the one hand, software could not wait for completion of hardware due to the schedule constraint. This required us to proceed with software design and development without access to a hardware target platform. Additionally, there was a requirement for building simulators and emulators for both development and testing. However, some of these tools, being built by Army engineers, would need to be certified before use and certification required running on hardware and software that was still under development.

These parallel development efforts would require a unique approach to development and risk management. At the macro or program level, we identified the following risk drivers.

1. **Schedule:** The schedule would be constrained and success-oriented, and the highest priority was placed on meeting schedule to allow for early fielding of the system. Time- and labor-intensive tasks such as documentation might have to be deferred until late in the schedule. In addition, many tasks that would normally be done sequentially would have to be done in parallel.
2. **Funding:** Limited funding was available for the software portion of the program. AdaRose would plan to make maximum use of available funding by multiple tasking of full-time engineers and by utilizing part-time

labor for engineering support elements such as configuration management (CM), quality assurance (QA), lab technicians, and network maintainers.

3. **Technical:** A number of engineering challenges were evident. The situational awareness (SA) computer had to be integrated to ensure that any SA failure would not impact the primary mission computer. Also, two third-party products – a radar measuring unit and a tactical communication module – needed to be integrated. AdaRose engineers had past familiarity with the tactical software, as well as prior experience with integrating situational awareness functionality and third-party products. Therefore, technical risk, although evident, was placed third in priority, as it did not appear to be a *showstopper* for the program.

Risk Scenarios

At the start of the program, we developed a number of risk scenarios to determine those events or trigger points we would have to watch, to warn us if and when the risk became imminent. Even though technical was not a serious risk driver, our No. 1 risk scenario involved the potential that the hardware, still in the design and development stage, would be substantially different from the specifications we were working from. If so, it could entail software rework and impact cost and schedule.

Our No. 1 concern was the communication interface between the tactical application and the inertial measurement/navigational unit. In the legacy system, this had been a straightforward Direct Memory Access (DMA) interface. Any change here was very risky because this unit was at the heart of the system and failure here meant the system could be dead in the water. The trigger point we watched for in this risk scenario was any change to that interface – and sure enough it occurred as the project evolved.

Due to hardware limitations on the tactical single-board computer, DMA could not be supported and the communication between the tactical application and the navigational unit had to be changed from DMA to an interrupt-driven serial connection. This, in turn, drove additional requirements to develop four new drivers to replace a single generic driver contained in the old architecture. This risk was mitigated somewhat by the fact that AdaRose engineers had prior formal training in developing software drivers for this operating system.

Controlling the Risks

As a baseline to accommodate top-level program visibility, AdaRose normally uses the typical Stair Step development process consisting of (1) requirements analysis (RA), (2) design, (3) code and unit test (CUT), (4) system level integration and test (SIT), and (5) formal qualification test (FQT). Then, depending on the type of software to be developed (e.g., new development, re-host, block update, prototype, etc.) and the constraints placed on the program (e.g., cost, schedule, technical), this baseline is modified/augmented for best program performance.

For this program, we decided to modify the baseline process with a spiral development approach to obtain maximum productivity from our developers and to mitigate major risk areas. At any point in time, programmers would be coding and unit testing in some areas while requirements analysis or design would be proceeding in others. We could also move out in those areas where the software was not yet dependent on hardware availability. For example, we decided, early on, to develop, application-specific, software simulators and communication protocol simulators to test the software – especially in those technical areas where rapid prototyping for proof-of-concept or early risk mitigation was warranted. The threads we used, throughout the development effort, to maintain coherency became known as *feature sets*. We found these to be invaluable risk mitigators.

> *"For this program, we decided to modify the baseline process with a spiral development approach to obtain maximum productivity from our developers and to mitigate major risk areas."*

Feature Sets

A feature set is a block of executable software that contains predefined features/ requirements that make up a subset of the entire program/application. A feature set can consist of nothing more than a rapid prototype to determine proof-of-concept, or a fully integrated and tested baseline. The purpose of feature sets is (1) to put before the user periodic drops of executable code to gain early concurrence and feedback of the included features/requirements; (2) to conduct early-on testing to reduce program risk and provide relatively *bug-free* software prior to entering FQT; and (3) to keep the development effort moving by allowing developers to move forward on those sets of features that are not dependent on other events, such as delivery of target hardware, special tools, or third-party products.

Ideally, as the program progresses and the software matures, periodic drops of feature sets would consist of the most current feature set along with all previous sets until such time that the final set is incorporated and the application is ready to enter FQT. Most of the early feature sets were tested using the developed software simulators.

The other *system level* risk mitigators that we used and that were described in my earlier articles on risk management [1, 2] are in the following sections.

Integrated Product Teams

Forming IPTs is another valuable approach to containing costs and reducing risks, especially those that might effect scheduling. The IPT facilitates problem solving, enables the team to rapidly respond to changing requirements, and prompts everyone to work on schedule.

Prototyping

Exploratory prototyping is an excellent risk mitigator if project requirements are ill-defined or likely to change before project completion. In addition, exploratory prototyping is an excellent way to clarify requirements, identify desirable features of the target system, and promote the discussion of alternative solutions.

Prototyping should answer two questions that are fundamental to software development and risk management: "Is the concept sound?" and "Is it worth proceeding further?" If the answer is not a clear *yes*, you may be setting yourself up for failure. More importantly, without this insight, you will give the customer a false sense of what can be accomplished. It is better to know this up front. Sometimes the most important risk management action you will take is to ask these fundamental questions.

As an example, on this project we needed to determine whether or not a viable software solution could be found

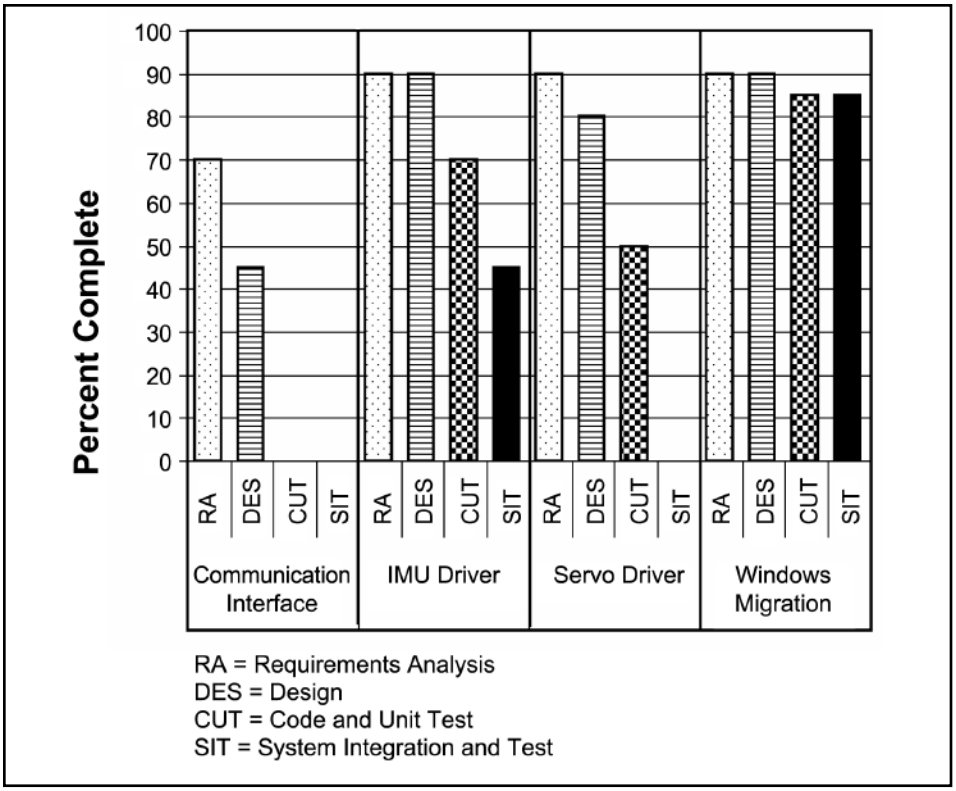

Figure 1: *Example of a Weekly Metric*

to replace the aging analog tachometers that controlled the rate of movement of the weapon system. We discovered that rate data was obtainable from the inertial measurement/navigational unit. We then proceeded to develop the prototype algorithms that substituted this rate data for the data from the tachometers. The next step was to *prove the concept*. This required just enough recoding to make it work on the existing system hardware. This was successful and as a result we were able to mitigate this risk early in the program.

If the answers and the risks are satisfactory in the exploratory prototyping phase, you can move on to evolutionary prototyping, which offers several benefits. It enables your team to quickly and efficiently build on proven aspects of the software. As a result, the core of the software's foundation is tested and proven early in the project, significantly reducing exposure to unknowns. It is an important contributor to *feature sets*.

Process Improvement

Improving processes should be ongoing throughout the project. For example, this project required a dual display mode on the operator's console. Rather than hold up development, while waiting on hardware to arrive, we did the necessary design and coding and used a dual monitor graphics card to test out and *prove* the design.

It is important to continually ask, "Is there a better way to get the job done?" Improving the way you do things cannot be done in a vacuum – communication at all levels is critical. Participate with your customers in IPTs and system management teams. In addition, be sure to meet with the teams' engineers on a regular basis for focused, but informal, discussions. While these meetings are exceptionally valuable, guard against extended meetings that cut into your teams' work time.

One alternative to lengthy meetings is to develop and distribute weekly status reports. These give each member insight into the progress of the entire project and a clear view of the big picture. Remember that you can have the best processes in place and still fail miserably in software development. A motivated, goal-oriented, and knowledgeable workforce will succeed even when the process is lacking. An example of one metric we used on a weekly basis is displayed in Figure 1.

Percent Complete

This metric provided top-level insight to the stage of development across blocks of functional requirements. We have shown here only four of the 13 major functional areas. Note that work in the communication interface area had not yet entered the code-and-unit test phase due to unavailability of hardware being developed by a third party. However, Windows migration was well ahead of the curve because it was not hardware dependent.

Also included in the weekly reports were more detailed descriptions of the major risk areas, for example see Figure 2.

Risk rankings were continuously re-evaluated and reprioritized throughout the program. As higher priority risks were worked off, others would move up to take their place. Risk mitigation became a dynamic real-time process.

Figure 2: *Example of Detailed Descriptions of a Major Risk Area*

IMU DRIVER

Complete, integrate, and deliver with Feature Set #7a.
Test the integrated build at the system level.

Risk Priority Ranking = 2
Schedule: Moderate/High
Technical: Moderate
Cost: Low/Moderate

Risk Mitigation: Develop dedicated IMU driver and modify application code IAW established DRU-H driver development plan. Apply best people. Utilize appropriate tools (SoftICE DDK). Develop software communication simulator. Use logic analyzer and oscilloscope for low-level debugging.

Action Officer(s): Primary – Tom; SW engineering – Frank and Ilya.
Suspense: IAW driver development plan, until removed from risk list.

Third Parties: A Mixed Blessing

If a product does fail, it is common for many developers to blame the project's failure on third parties. In some cases they are correct. At times you will have no choice but to elicit their help. The key is to minimize how much you depend on them.

Any time you rely on a product or service from someone outside of your group your risk of failure or delay increases. Your team may do everything right, but if a crucial third party does not, your work may be

in vain. To illustrate this, consider the risks you assume by depending on three crucial components of your project from start to finish. Assume each product has an 80 percent chance of arriving on time and fully functional. The probability of success for all three combined is not 80 percent, it is 0.80 x 0.80 x 0.80 or 51 percent. In other words, your project now has only a 50-50 chance of success. Do not assume that third parties will have the same priorities that you have. Use daily communication with them to keep them in the loop and make them a part of your team.

System-Level Cohesion

Needless to say, software cannot be developed in a vacuum. In the ideal software world, we would hope to have qualified hardware, emulators/simulators, and all design and interface documents delivered at project start. But that is not realistic, especially with military tactical systems. We find that software and hardware are an integral, non-separable entity. Quite often participants in *system* development efforts will *finger point* and blame the other guy for lack of progress.

On this project, we all realized the many challenges and knew that a successful outcome depended on a strong team effort. As a result, we witnessed almost daily instances of engineers supporting each other – often putting aside their own work to help move forward a higher priority effort. We saw a close working relationship develop between our software developers and the hardware developer. AdaRose engineers quite often diagnosed hardware anomalies and provided workable solutions. At the IPT level, all were well aware of the risks and a *helping hand* rather than a *pushing hand* was the norm.

Results

As of this writing, the project is midway through formal qualification test. The schedule is still paramount but software development was able to proceed in advance of hardware availability by identifying and mitigating those critical risk areas that could be worked off early. We did this, up front, through rapid prototyping and by providing feature sets to show proof-of-concept and provide executable code to qualify the hardware and help certify the simulation/emulation tools. This required a tailoring of our process and maintaining a viable and dynamic risk management program at the system level.

Summary

Software development will always include risks, but none are insurmountable if you are prepared to face them at the start. Risk management is an excellent way to prepare for daily challenges. Risk management must not only be implemented but continually reassessed throughout the life of the project. Do not blindly follow any particular process but do tailor your process to the job at hand.

A viable risk management plan can mean the difference between success and failure. It should, above all else, be flexible and encourage initiative. Remember to always look ahead, use rapid prototyping if necessary, develop simulators if necessary, follow a defined program to minimize and manage risks, use a good set of metrics, keep the customer in the loop, and always follow the fundamentals of sound application development. Following this risk management approach will not guarantee excellent software development, but over time it will certainly contribute to your success.◆

References

1. Holt, George. "Risk Management Fundamentals in Software Development." CROSSTALK, Aug. 2000: 12-14 <www.stsc.hill.af.mil/crosstalk/2000/08/holt.html>.
2. Holt, George. "Software Risk Management – The Practical Approach." Software Tech News 2.2 <www.softwaretechnews.com/technews 2-2/practical.html>.

About the Author

George Holt is president and chief executive officer of AdaRose Inc. He has a wealth of program management experience primarily with military tactical systems. AdaRose recently re-hosted 230,000 lines of Army tactical software, written in Ada, to a digital control unit containing three single-board computers, providing navigation, artillery fire control, situational awareness, and a prognostic/diagnostic capability. Holt is the author of many technical publications and co-author of "Strategy: A Reader."

AdaRose Inc.
430 Marrett RD
Lexington, MA 02421
Phone: (802) 728-9448
Fax: (781) 274-7359
E-mail: holt.adarose@verizon.net

Coming Events

March 5-12
IEEE Aerospace Conference
Big Sky, MT
www.aeroconf.org

March 7-10
SEPG 2005
Seattle, WA
www.sei.cmu.edu/sepg

March 15-16
Dayton Information Security Conference '05
Dayton, OH
www.gdita.org/inc/eventdetail.asp?eventID=340

April 4-6
DTIC Annual Users Meeting and Training Conference
Alexandria, VA
www.dtic.mil/dtic/annualconf

April 5-7
Federal Office Systems Exposition (FOSE) 2005
Washington, DC
www.fose.com

April 18-21
2005 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org

May 2-6
Practical Software Quality and Testing (PSQT) 2005
Las Vegas, NV
www.qualityconferences.com

May 8-12
Nano Science and Technology Institute 2005 Conference
Anaheim, CA
www.nanotech2005.com/